Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2013-031 December 29, 2013

# Synthesis of Randomized Accuracy-Aware Map-Fold Programs

Sasa Misailovic and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# Synthesis of Randomized Accuracy-Aware Map-Fold Programs

Sasa Misailovic and Martin Rinard
MIT CSAIL
{misailo,rinard}@mit.edu

**Abstract.** We present Syndy, a technique for automatically synthesizing randomized map/fold computations that trade accuracy for performance. Given a specification of a fully accurate computation, Syndy automatically synthesizes approximate implementations of map and fold tasks, explores the approximate computation space that these approximations induce, and derives an accuracy versus performance tradeoff curve that characterizes the explored space. Each point on the curve corresponds to an approximate randomized program configuration that realizes the probabilistic error and time bounds associated with that point.

## 1 Introduction

Many computations exhibit inherent tradeoffs between the accuracy of the results they produce and the time required to produce these results. Examples include audio and video processing applications, numerical computations, machine learning, and search applications. A key aspect in tuning these applications is finding alternative implementations of the program's subcomputations and configurations of these subcomputations that yield a profitable tradeoff between accuracy and performance.

Researchers have previously proposed several language-based techniques for navigating the tradeoff space and identifying or synthesizing alternative versions of subcomputations that help deliver desired tradeoffs. Many of these approaches are empirical in nature – to find profitable tradeoffs they execute programs on a set of representative training inputs and use this information to make decisions when faced with (previously unseen) production inputs [16,17,2,14,3,12,19]. In contrast to these dynamic approaches, static techniques [13,8,21,6] allow a user to specify the properties of program inputs (e.g., specific ranges or probability distributions of inputs) and computations (e.g., continuity), which can be used to guide the optimization of the program. Out of these static approaches, only the analysis presented in [21] is used to reason about accuracy versus performance tradeoffs.

**Syndy.** This paper presents Syndy, a technique for accuracy-aware optimization of approximate map-fold computations. Syndy takes as input a program that implements a fully accurate map-fold computation, a set of intervals of the inputs of this computation, and a set of alternative implementations of program's subcomputations. Syndy produces as output a transformed, randomized, approximate program that can execute at a number of points in the program's underlying accuracy versus performance tradeoff space. This tradeoff space is induced by the transformations (specifically, subcomputation substitution and sampling of fold operands) that Syndy implements.

Syndy uses an optimization algorithm presented previously in [21] to explore the tradeoff space and construct an optimal tradeoff curve of the approximate program. This optimization algorithm defines and solves mathematical optimization problems to construct a tradeoff curve that contains optimal tradeoffs between accuracy and performance for map-fold computations expressed in an abstract model of computation based on graphs. Each point on the tradeoff curve contains a set of randomized program configurations that deliver the specified accuracy/performance tradeoff.

**Contributions.** This paper presents a programming language for map-fold computations and an optimization algorithm that operates on programs written in this language. Specifically, in comparison with previous research [21], this paper makes the following contributions:

- **Language.** It presents a language for expressing map-fold computations and defines the semantics of this language ([21], in contrast, worked with an abstract model of computation based on graphs).
- **Synthesis of Alternate Computations.** It presents program transformations that induce the underlying accuracy versus performance tradeoff space ([21], in contrast, worked directly on the graphs from the abstract model of computation).
- **Mapping.** It presents a mapping from the Syndy programming language to the abstract model of computation from [21]. This mapping enables the use of the optimization framework to find optimal tradeoffs for map-fold programs.
- **Implementation and Evaluation.** It presents experimental results from an implementation of the optimization algorithm ([21], in contrast, presented a theoretical algorithm with no implementation).

## 2 Example

Figure 1 presents an example map-fold program. The program takes as input features of images (such as edges of objects in the image and the foreground/background data) and a set of models that represent the position of persons in the image. The program analyzes the models and computes the maximum likelihood, which corresponds to the model with minimum error from the image data. This computation is derived from a part of the Bodytrack motion tracking application [5] that computes the location of persons in a sequence of images from security cameras.

**Functions.** Syndy allows specifying two kinds of functions – primitive and composite. *Primitive functions* are defined in an external programming language and can use arbitrary complex language constructs. Syndy treats the primitive functions as black box computation. *Composite functions* are defined as expressions in Syndy's language. Composite functions call and operate on the results of the primitive functions. Both primitive and composite functions produce numerical results.

The functions `ImageEdge` and `ImageInside` are primitive. These functions take two input parameters, `image` (image pixel information) and `model` (the model of the person's location), and produce numerical values as the outputs. The function `ImageEdge`

```
function ImageEdge(image, model);
function ImageInside(image, model);
function Exp(val);
function Max(val1, val2);

function Likelihood(image, model) :=
    Exp(-1*((ImageEdge(image, model)
        + ImageInside(image, model)))

program (images, models) :=
    fold(0, Max,
        map (Likelihood,
            zip(images, models)))
```

Fig. 1: Example Program

```
SpecF(ImageEdge) := (
    { (IEdge, 0, T_e0), (IEdge1, E_e1, T_e1),
      (IEdge2, E_e2, T_e2), (IEdge3, E_e3, T_e3)
    }, (NA, NA))
SpecF(ImageInside) := (
    { (IInside, 0, T_i0), (IInside1, E_i1, T_i1),
      (IInside2, E_i2, T_i2), (IInside3, E_i3, T_i3)
    }, (NA, NA))
SpecF(Exp) := ({ (Exp, 0, T_exp) }, 1)
SpecF(Max) := ({ (Max, 0, T_max) }, (1,1))

SpecI(images) := (400)
SpecI(models) := (400)
```

Fig. 2: Function And Input Specifications

compares the prediction of the person's location from the model with the sharpness of image object edges. The function `ImageInside` compares the model prediction with the foreground/background surfaces from the image. The result of each function is the error between the image data and the model, which is a value between 0 and 1.

The function `Likelihood` is composite – it computes the arithmetic expression that calls three primitive functions, `Exp` (exponentiation), `ImageEdge`, and `ImageInside`.

**Program Inputs.** The program takes as input the list containing pointers to the raw image (`image`) and the list of models (`models`). Both lists contain *structurally complex* data structures. These data structures can only be processed by primitive functions; the map-fold computation operates on numerical data and only passes the complex data structures to primitive functions.

**Main Computation.** The computation uses the helper *zip* operator to combine the two input lists and construct a list of pairs of the images and models. This list is passed as input of the map operator. The map operator applies the function `Likelihood` for each element of the input list. Since each element of the input list is a pair of values, the map operator unpacks the pair elements and passes them as arguments of the function `Likelihood`. The map operator uses a *lazy evaluation* strategy: its output list contains the expression terms that evaluate to the likelihood for each input parameters. When the result of the map operator is required, these expressions are evaluated to produce the numerical values.

The fold operator computes the maximum value of the likelihoods of all models in the input list using the built-in `Max` function. Since the input list of the fold operator contains the expressions that compute likelihoods, these expressions are evaluated before executing the `Max` operation. `Max` then performs comparisons on numerical values. The result of the fold operator is a numerical value that is the maximum likelihood. This is also the result of the program.

### 2.1 Approximating Computations

This computation has several opportunities for trading accuracy for additional performance. First, the functions `ImageEdge` and `ImageInside` in the ideally accurate implementation compare all pixels of the input image with the corresponding model location. Approximate implementations of these functions can sample only a subset of pixels when computing the image/model difference. Second, the maximization fold operator

may skip some of its inputs, effectively looking for the maximum likelihood of only a subset of models (typically without incurring a major accuracy penalty). When the fold operator skips likelihoods computed from some of the models, then the previous map operator's computation for these models can also be skipped; this computation skipping may provide significant additional performance benefits. Syndy uses specifications of primitive functions and inputs to generate alternative implementations of the computation and exploit these approximation opportunities.

**Accuracy/Performance Specifications.** The developer provides specifications of accuracy and performance of alternative function implementations. Figure 2 presents the specification of the example functions (**SpecF**). The specification consists of two parts. The first part is a set of alternative function implementation specifications. Each specification contains the name of function, the expected absolute error incurred by the execution of this implementation, and the expected execution time.

The function `ImageEdge` has four implementations: the original implementation (denoted as `IEdge`) does not incur any error, and it executes in time $T_{e0}$. The other three implementations are approximate; each of these approximate implementations performs a regular sampling of the image pixels. The error that the approximate implementations incur is greater than 0, but their execution time is smaller. Both error and time specifications are numerical constants – for instance, they may have the following values: `(IEdge, 0.000, 0.174)`, `(IEdge1, 0.004, 0.097)`, `(IEdge2, 0.012, 0.059)` and `(IEdge3, 0.016, 0.051)`. The specification for `ImageInside` is similar. The functions `Exp` and `Max` have only a single, fully accurate implementation.

The second part of the accuracy/performance specification of the function is the *sensitivity* of the function's result to the changes in each of its numerical input parameters.[1] Consider the function `Max`. Its sensitivity vector has two elements (since the function has two parameters). Both sensitivity coefficients are 1, which indicates that the function `Max` does not amplify the error of its arguments. Therefore, the noise of the output of the maximum operation is proportional to the noise introduced in each of the function's arguments. The developer has specified that the sensitivity of the function `Exp` is 1. For this the developer uses the additional information that the functions `ImageEdge` and `ImageInside` produce a result between 0 and 1. Finally, the functions `ImageEdge` and `ImageInside` take as input complex data structures. Since the sensitivity is defined only for numerical parameters, the developer uses the keyword *NA* to denote that the sensitivity information is not applicable for these parameters

**Input Property Specification.** The function **SpecI** specifies the properties of the inputs of the computation, such as the size of the input lists and the intervals of the input values (if applicable). The example specification specifies that the lists `images` and `models` have 400 elements each. For lists of numerical data, a user can also specify the intervals of the inputs; however since the input lists in the example contain complex data structures, this part of the specification is not applicable.

[1] Specifically, the sensitivity coefficients are Lipschitz constants of a function. For instance, for a function with one argument, $S$ is a Lipschitz constant if $\forall x \,.\, |f(x + \delta x) - f(x)| \leq S \cdot |\delta x|$. The definition of sensitivity can also be restricted for $x$ that belongs to a closed subinterval of numbers.

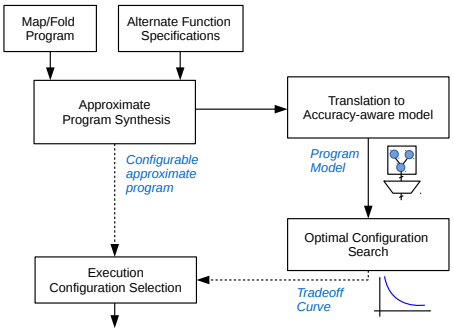


Fig. 3: Synthesis Framework Overview

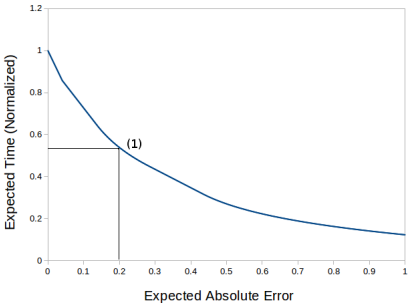


Fig. 4: Bodytrack Tradeoffs

## 2.2 Automating Transformations

Figure 3 presents an overview of the framework that synthesizes configurable approximate implementations of map-fold computations and searches for configurations of these computations that deliver optimal tradeoffs between performance and accuracy of these computations. The framework takes as input the original map-fold computation and the function and input specifications. The framework consists of 1) the program synthesis component, which constructs an approximate configurable program from the original one, and 2) the optimal configuration search component, which computes configurations that deliver optimal tradeoffs between accuracy and performance.

**Approximate Program Synthesis.** We define the semantics of the Syndy's language of map-fold computations that allows for effectively using the probabilistic choice expression to implement the choice between multiple function implementations and sampling of inputs of fold operations. The probabilistic choice expression, $e_1 \ {}_q\oplus \ e_2$, is controlled by the configuration variable $q$: the expression $e_1$ is executed with probability $q$ and the expression $e_2$ is executed with probability $1 - q$. The value of $q$ is specified at the beginning of the program execution.

Syndy synthesizes an approximate configurable implementation of the function by replacing the function calls in the original program with the call to the synthesized function that contains the probabilistic choice between the original and all alternative implementations of the function (declared in the function specification). For example, the configurable implementation of the function `Likelihood` is

```
  ImageEdge'(image, model) := IEdge(image, model)  qe1⊕ IEdge1(image, model)  qe2⊕
                              IEdge2(image, model)  qe3⊕ IEdge3(image, model)
ImageInside'(image, model) := ImageInside(image, model)  qi1⊕ IInside1(image, model)  qi2⊕
                              ImageInside2(image, model)  qi3⊕ IInside3(image, model)

Likelihood'(image, model) := Exp(-1*(ImageEdge'(image, model) + ImageInside'(image, model))).
```

This alternative computation randomly calls the original or approximate implementations of the functions `ImageEdge` and `ImageInside`. Probabilistic choice operations

execute from left to right. The variables $q_{e1}, q_{e2}, q_{e3}$ and $q_{i1}, q_{i2}, q_{i3}$ control the execution of the function `Likelihood'`. Syndy changes the map operator to call the function `Likelihood'` instead of the original function `Likelihood`.

The sampling of the maximization node is performed similarly. The function `Max`$(a, b)$, which computes the maximum between the temporary aggregation variable $a$ (which contains the maximum value seen so far) and the new input $b$, is replaced with

$$\texttt{Max(a,b)} := \texttt{Max(a,b)}\ {}_{q_m}\oplus\ \texttt{a}.$$

This alternative implementation therefore skips the calculation of the subcomputation that computes the input $b$ with probability $1 - q_m$. Therefore, the final approximate configurable program becomes:

```
program (images, models) :=
    fold(0, Max', map(Likelihood', zip(images, models)))
```

### 2.3 Optimal Configuration Synthesis

The variables $q_{e1}$, $q_{e2}$, $q_{e3}$, $q_{i1}, q_{i2}, q_{i3}$, and $q_m$ represent the *configuration* of the synthesized approximate program. This program can execute at many points in the tradeoff space by selecting the values of the configuration variables. To synthesize the configurations of the approximate map-fold program, Syndy 1) translates the map-fold computation into an equivalent graph model from the accuracy-aware model of computation and 2) runs the optimization algorithm on this graph model to obtain the program configurations.

**Accuracy-aware Model of Computation.** The accuracy-aware model of computation presented in [21] is a tree of computation and reduction nodes. A *computation node* represents the computation that executes independently on $m$ inputs in parallel. The number of inputs $m$ represents *multiplicity* of the node. The computation performed on each input is represented as a dataflow *directed acyclic graph*, where each node represents a computation (called *function nodes*), and each edge between nodes represents a data flow between the function nodes. A *reduction node* represents aggregate operations on $m$ inputs. The aggregate operations that the model supports include summation, averaging, minimization and maximization.

**Translation to Accuracy-aware Model.** Syndy translates the map-fold program to a tree of computation and reduction nodes. The functions with alternative implementations in map-fold programs are translated to function nodes. The map operators are translated to computation nodes. The fold operators with appropriate function (sum, avg, min, or max) are translated to the corresponding reduction node. The translation component also represents each alternate synthesized computation as a transformation within the accuracy-aware model. We discuss this translation in Section 5.

**Optimal Configuration Search Algorithm.** The configuration optimization algorithm uses the graph model to search the tradeoff space induced by the program configurations. The optimization algorithm produces an optimal accuracy/performance *tradeoff curve* [21]. Each point on the tradeoff curve contains 1) the upper bound on the expected absolute error, 2) the upper bound on the execution time, and 3) the configuration of the

transformed program that delivers the specified tradeoff. Figure 4 presents the optimal tradeoff curve between the error and the execution time that Syndy produces for the example computation. The execution time is normalized to the time of the original (fully accurate) implementation of the computation. We discuss the implementation and the initial evaluation of the configuration optimization algorithm in Section 6.

### 2.4 Program Configuration Selection

At the start of the execution of the synthesized randomized program, the runtime system selects the program's configuration based on the user-provided error tolerance bound $\Delta$. The runtime reads from the tradeoff curve the configuration that guarantees that the execution error is smaller than $\Delta$. For instance, if the user selects tolerance bound $\Delta = 0.2$, the runtime will select the configuration from the point *(1)* in Figure 4. This configuration executes in time at most 55% of the execution time of the original (non-approximate) program. The runtime system replaces each configuration parameter $q$ with the corresponding numerical value from the obtained configuration vector. Then such program is guaranteed to produce the result that is within the user-specified bound.

## 3 Language

Figure 5 presents the syntax of the language of map/fold computations. A program is a set of function declarations and definitions ($D$) and a program expression that consists of **map** and **fold** operators. A developer can declare *primitive* functions, which are external black box computations, or define *composite* functions.

**Data Types.** The program operates on the following inputs:

- *Numerical Values.* The numeric set represents rational numbers. The range of numbers that this type represents is not bounded.
- *Opaque Values.* The inputs of the program may also contain elements of complex data types. These data types are opaque to the Syndy's computation. Opaque values can only be passed as arguments of primitive functions.
- *Lists.* An input list contains numeric or opaque values. Each input list has a (known) finite size.

**Functions.** The language supports two kinds of functions:

- *Primitive functions.* These are atomic black box computations (from the perspective of Syndy's language). These functions can be implemented in an external programming language. Primitive functions can take numerical or complex data structures as input, but they produce only numerical values as outputs. A primitive function is declared by specifying its name and the list of parameters.
- *Composite functions.* Composite functions are defined in Syndy 's language. The function declaration includes the function name and the list of formal parameters. The body of these functions is an expression that produces a numerical value.

Both primitive and composite functions return numerical values. We note that only primitive functions can operate on vales of opaque type. We omit the presentation of a type analysis that ensures that the primitive functions accept the parameters of the proper type (i.e., numerical or opaque) and that the arithmetic operations operate on numerical data.

$$
\begin{array}{lll}
n \in \text{Numeric} & \text{D} \in \text{Decl} \to f(x_1, \ldots, x_n) := e \mid & e \in \text{Exp} \to x \mid e_1\ op\ e_2 \mid \\
o \in \text{Opaque} & \qquad\qquad f(x_1, \ldots, x_n); & \qquad\quad e_1\ {}_p\!\oplus\ e_2 \mid \\
x \in \text{Vars} & \text{L} \in \text{LExp} \to a \mid [t_1, \ldots, t_k] \mid \mathbf{map}(f, \text{L}) \mid & \qquad\quad \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \\
a \in \text{ListVars} & \qquad\qquad [t\langle x\rangle : x\ \mathbf{in}\ 1 : n] \mid \mathbf{zip}(L_1, L_2) & \qquad\quad f(e_1, \ldots, e_k) \mid \bar{e} \\
p \in [0, 1] & \qquad\quad t \to \bar{e} \mid (\bar{e}, t) & \bar{e} \to v \mid \mathbf{fold}(n, f, \text{L}) \\
 & & v \to n \mid o
\end{array}
$$

$$
\mathcal{P} \in \text{Prog} \to \text{D}^*\ \mathbf{program}(a^*) := [\bar{e} \mid \text{L}]
$$

Fig. 5: Syndy's Target Language Syntax

**Probabilistic Choice Operation.** The language supports the probabilistic choice operation ${}_p\!\oplus$, which computes the result of the expression $e_1$ with probability $p$; it computes the result of the expression $e_2$ with probability $1 - p$. The choice is controlled by the numerical constant $p \in [0, 1]$.

**Arithmetic Operations.** The language supports the standard arithmetic operations, including addition, subtraction, and multiplication.

**Tuples.** The language defines tuples as an auxiliary data structure that allows the developer to use functions with multiple arguments within the **map** operator. It also defines the auxiliary **zip** operator, which produces a list of pairs from two lists of the same size. We define an additional syntax construct $(\bar{e}_1, \bar{e}_2, \ldots, \bar{e}_n)$ to succinctly represent the tuple $(\bar{e}_1, (\bar{e}_2, \ldots, \bar{e}_n))$ and $\mathbf{zip}_n(L_1, L_2, \ldots, L_n)$ to represent $\mathbf{zip}(L_1, \mathbf{zip}(L_2, \ldots, L_n))$.

**Map Operators.** A **map** operator takes as input a function $f$ and a list of inputs. It applies the function $f$ independently on each element of the input list $L$ to produce the list of outputs. An input list of a map operator is either a program input or an enumerated list of the results of the previous subcomputations, $[t_1, \ldots, t_n]$.
We define a *parameterized enumerated list*, an additional language construct $[t\langle x\rangle : x\ \mathbf{in}\ 1 : n]$, to succinctly represent the enumerated list $[t[1/x], \ldots, t[n/x]]$, in which all free occurrences of the variable $x$ in all elements of $t$ are substituted by a constant from 1 to $n$, respectively.

**Fold Operator.** A **fold** operator takes as input 1) a starting numerical value $n$, 1) a function $f$, which takes as input a temporary value of the accumulator and a single element from the list and produces the results of aggregation, and 3) a list of numerical inputs. In each step, the function $f$ reads a single input from the list and computes the intermediate aggregate value. The output of the fold operator is a single numerical value that aggregates the contributions of all elements of the input list.

**Well-Formed Program.** The root expression of a well-formed program is a map or a fold operator. All tuples within an enumerated list in a well-formed program have the same number of elements and all lists referenced by a zip operator have the same length.

### 3.1 Language Semantics

We first note several points that have influenced the design of the language:

– *Probabilistic Randomized Execution.* The language supports a probabilistic choice operator, which (with specified probability) executes one of its argument expressions. We use this operator as a foundation for expressing randomized transformations, such as implementation substitution and sampling, within the language.
– *Lazy Evaluation of Map Operators.* To support sampling of fold operators, the semantics of the language delays the execution of the computation of the map operators until these results are needed. Therefore, if some of the results of the map operator are not used by the subsequent computation, then these results are not computed.
– *Isolation of External Computation.* To support a broader set of computations, the language enables calling an arbitrary complex computation (written in an external programming language) that operates on individual pieces of data. However, the only effect of the external computation is the result that it returns; the external computation does not have visible side effects.

**Preliminaries.** The semantics of the language models an abstract machine with a memory that contains the inputs of the program. The memory is read-only. To represent the state of the computation, we define the set of generalized expressions GExp = Exp ∪ LExp. The canonical expressions GVal that the abstract machine produces are numerical and opaque values, and tuples and lists of values and functions applied to closed expression terms.

Figure 6 presents the transition rules of the operational semantics. These rules represent the program execution with a probabilistic, non-deterministic transition system. The big-step evaluation transition is a ternary relation $\cdot \stackrel{\cdot}{\Longrightarrow} \cdot \subseteq \text{GExp} \times [0,1] \times \text{GVal}$. The probabilistic big-step evaluation relation $\mathbf{E} \stackrel{p}{\Longrightarrow} \mathbf{v}$ states that the expression $\mathbf{E} \in$ GExp evaluates in one or multiple steps to the final canonical expression $\mathbf{v} \in$ GVal with probability $p$. A numerical or opaque value always evaluates to itself with probability 1.

We define a substitution operation $e_1[e_2/x]$ in a standard way – it produces a new expression obtained by syntactically replacing all free occurrences of the variable $x$ within the expression $e_1$ with another expression $e_2$. Multiple substitution operations are applied from left to right.

**Probabilistic Choice Operation.** The rule for probabilistic choice evaluates to the expression $e_1$ with probability $p$ or the expression $e_2$ with probability $1 - p$. The rule specifies that only one of the expressions $e_1$ or $e_2$ will be computed.

**Arithmetic Operations.** The rules the arithmetical expressions are standard: the probability that an arithmetic expression will produce a result $n_1$ *op* $n_2$ is equal to the product of probabilities that the expressions $e_1$ and $e_2$ evaluate to the numerical values $n_1$ and $n_2$, respectively.

**Primitive Function Calls.** The transition rule for a primitive function call E-FUN-EXT first evaluates all function's parameter expressions in a *call-by-value* fashion. Then, the opaque values and the computed numerical values are passed to the primitive functions. The result of a primitive function $n_f$ is the only observable effect of the function's execution. This execution does not have to be deterministic; therefore it may produce a specific result $n_f$ with probability $p_f$. For simplicity, we assume that primitive function calls terminate for all inputs, i.e., $\sum p_f = 1$.

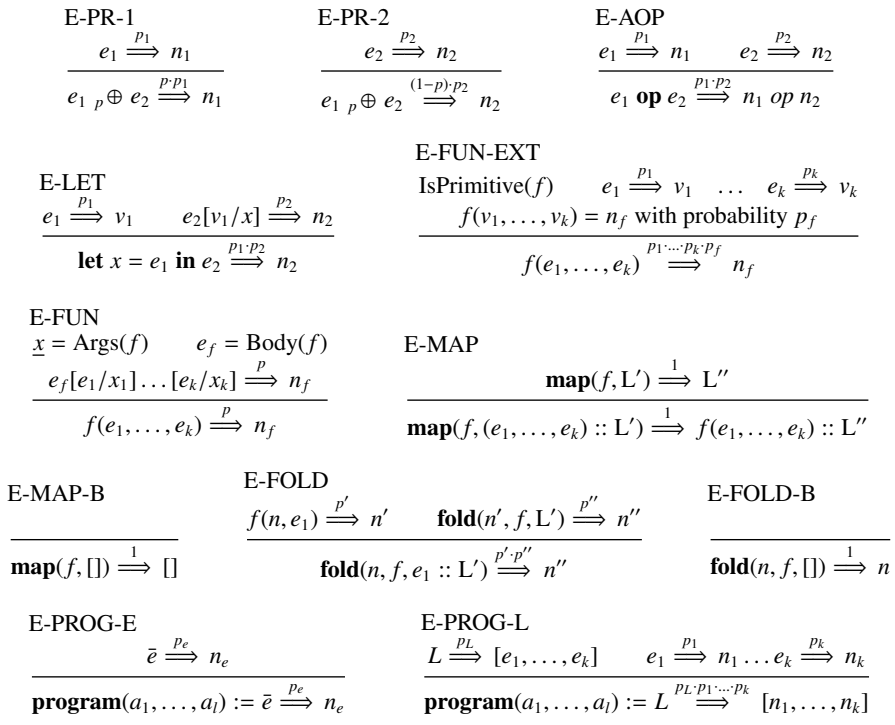


Fig. 6: Dynamic Semantics Transition Rules

**Composite Function Calls.** The transition rule for a composite function call E-FUN uses the *call-by-name* evaluation strategy. The arguments of the function are computed only when needed during the execution of the function's body. Therefore, if some function's arguments are not required, for example due to the execution of the probabilistic choice within the function's body, then the expression that computes the argument's value is never executed.

**Let Expression.** The let expression allows specifying an intermediate variable name $x$ to represent the result of some other expression. The expression $e_1$ can 1) be an opaque value or 2) evaluate (eagerly) to a numerical expression. The resulting value is then substituted in the expression $e_2$.

**Map Operator.** The rule E-MAP states that in a single step the **map** operator applies a composite function $f$ to the first element from the list L. The first element can be a single expression or a tuple of expressions. The rule treats each expression as an argument of the function $f$. The evaluation of the function $f$ is deferred until it is required by the subsequent computation. The operator :: is used to concatenate the expressions to produce the output list.

**Fold Operator.** The rule E-FOLD first obtains the previously computed value $n$ and the first element from the list of inputs $L_1$ and applies the function $f$. The execution of the function $f$ produces the result $n'$ with probability $p$. The computed value $n'$ is passed as the initial value of the next step of the **fold** operator, which executes on the remainder of the input list and produces a final numerical result.

**Program-Level Semantics.** The top-level expression of the body of the program can be a fold or a map operator. The rule E-PROG-E specifies that the result of the program is equal to the numerical value that the fold operator produces. The rule E-PROG-L specifies that the result of the map operator (which is a list of closed expression terms) evaluates to a list of numerical values; this list is returned by the program expression.

### 3.2 Expression Expectation

To define the probability distribution of the results produced by the map-fold computation, we first define the set of final values and probabilities to which a closed expression term evaluates, $\mathrm{K}(e) = \{(n, p) : e \overset{p}{\Longrightarrow} n\}$. The induced probability mass function is then $\mathbb{P}_{\mathrm{K}(e)}(n) = p$. This distribution is discrete under the condition that the sets of the opaque values are countable. Note that the set Numeric is by definition countable. We will assume this condition in the remainder of the paper.

**Definition 1 (Expectation).** *Let* $\mathrm{K}(e) = \{(n, p) : e \overset{p}{\Longrightarrow} n\}$ *be the set of values that the closed expression term e evaluates to. The expected value of the expression e is* $\mathbb{E}[\, e \,] = \sum_{(n,p)\in \mathrm{K}(e)} p \cdot n.$

The expected value of a numerical expression is the weighted sum of all the values that the expression can evaluate to.

**Definition 2 (Conditional Expectation).** *Let* $e_1 \overset{p_1}{\Longrightarrow} n_1$ *and* $e_2 \overset{p_2}{\Longrightarrow} n_2$*, and let* $g(e_1, e_2)$ *be some function of the closed expressions* $e_1$ *and* $e_2$*. Then, the conditional expectation of g under the condition that* $e_2$ *evaluated to* $n_2$ *(expressed as* $e_2 \dot{\Longrightarrow} n_2$*) is equal to* $\mathbb{E}[\, g(e_1, e_2) \mid e_2 \dot{\Longrightarrow} n_2 \,] = \sum_{(n_1,p_1)\in \mathrm{K}(e_1)} p_1 \cdot g(n_1, n_2).$

The expected value of $g(e_1, e_2)$ conditioned on the observation that $e_2$ evaluated to $n_2$ is equal to the expectation of $g(e_1, n_2)$.

**Lemma 1 (Total Expectation).** *Let g be an auxiliary function and* $e_1$ *and* $e_2$ *two closed expressions. Then,* $\mathbb{E}[\, g(e_1, e_2) \,] = \mathbb{E}[\, \mathbb{E}[\, g(e_1, n_2) \mid e_2 \dot{\Longrightarrow} n_2 \,] \,]$

### 3.3 Specification of Alternative Primitive Functions

The specification of a primitive function $f$ is a tuple $(f, \mathrm{A}, \mathrm{S})$. It contains the unique name of the function, $f$, the set of alternative implementations of the function, A, and the vector of sensitivities of the function's outputs to the perturbation of each input of the function, S.

Each *alternative* implementation of the function is a less accurate (but potentially faster) version of the computation performed by the *original* implementation $f$. An alternative implementation is specified by a tuple $(f', E_{f'}, T_{f'})$. Its first element is the name of the alternative implementation, the second is the upper bound on the expected absolute error $\mathrm{E}_f$, and the third is the expected execution time of the function, $T_f$. As a special case, the specification of the original (fully accurate function) is $(f_0, 0, T_{f0})$.

The expected error and sensitivities of the function can be defined in terms of the dynamic semantics of the program. We will first focus on the functions that operate only on numerical inputs.

**Expected Error Specification.** Let $f$ be the original and $f'$ the alternative implementation of a primitive function. Furthermore, let $x_1 \in I_{f,1}, \ldots, x_k \in I_{f,k}$ be the inputs of the function $f$, bounded on the intervals $I_{f,1}, \ldots, I_{f,k}$. Then,

$$E_{f'} := \mathbb{E}[\,|f(n_1, \ldots, n_k) - f'(n_1, \ldots, n_k)|\,], \tag{1}$$

which is valid for all $n_1 \in I_{f,1}, \ldots, n_k \in I_{f,k}$. The expectation is taken over the inherent randomness of the underlying computation; the expected error is not a function of the inputs of the function. Note that the intervals of the inputs are part of the input specification. The intervals of the intermediate inputs/outputs are computed by an external interval analysis.

**Sensitivity Specification.** While the expected error represents the error that emerges in the computation, the sensitivity quantifies and bounds the error originating from the computation that preceded the current function. Let $S_i$ be the sensitivity index of the function's $i$-th input. For all inputs $n_1, \ldots, n_k$ and $\hat{n}_1, \ldots, \hat{n}_k$ that lie in the intervals $I_{f,1}, \ldots, I_{f,k}$, the sensitivity indices are the minimum bound on the absolute difference $|f(n_1, \ldots, n_k) - f(\hat{n}_1, \ldots, \hat{n}_k)| \leq \sum_{i=1}^{k} \mathrm{S}_i |n_i - \hat{n}_i|$,.

If we replace the numerical values with expressions that compute them, then (using Lemma 1) the expression for the expected propagated error becomes

$$\mathbb{E}[\,|f(e_1, \ldots, e_k) - f(\hat{e}_1, \ldots, \hat{e}_k)|\,] \leq \sum_{i=1}^{k} \mathrm{S}_i \mathbb{E}[\,|e_i - \hat{e}_i|\,]. \tag{2}$$

This bound is valid for all inputs within the specified input intervals.

Finally, the bound on the total expected error (from both locally induced and propagated error) is

$$\mathbb{E}[\,|f(e_1, \ldots, e_k) - f'(\hat{e}_1, \ldots, \hat{e}_k)|\,] \leq E_{f'} + \sum_{i=1}^{k} \mathrm{S}_i \mathbb{E}[\,|e_i - \hat{e}_i|\,]. \tag{3}$$

**Opaque Typed Parameters.** If some of the arguments of the function $f$ are opaque values, then the previous expectation expressions are required to be valid for all opaque value arguments in Opaque. The sensitivity of opaque value arguments is not defined. A developer can write a special value *NA* to denote unknown sensitivities of opaque arguments.

### 3.4 Performance Model

The computation's performance model approximates the total execution time of the computation by defining the base expected execution times for expressions and function calls. The execution time function $\mathrm{T}\quad(e)$ returns the expected execution time of the expression $e$. The expectation in this case is over the uncertainty caused by the variability of the underlying concrete hardware platform.

This model operates under the assumption that the majority of the program's execution time is spent in the primitive functions. Therefore the execution time of simple operations is negligible. The execution time of a binary expression is $\mathrm{T}\quad(e_1\ op\ e_2)$ as $\mathrm{T}\quad(e_1)+\mathrm{T}\quad(e_2)$. The execution time of a probabilistic expression is $\mathrm{T}\quad(e_1\ {}_p\oplus e_2) = p\cdot\mathrm{T}\quad(e_1)+(1-p)\cdot\mathrm{T}\quad(e_2)+\tau_\oplus$, where $\tau_\oplus$ is the (non-negligible) time required to produce one random sample.

The execution time of each primitive function is defined by its specification. This execution time does not depend on the arguments of the function. The execution time of a composite function call is equal to the execution time of its body (with replaced expressions for arguments).

The map operator does not directly evaluate the result of its function $f_{elem}$ on the input list; it merely produces an output list, which is evaluated by the surrounding fold expression or by the program expression. If its result expression is computed by the top level program expression, then the program expression's execution time is equal to $m\cdot\mathrm{T}\quad(f_{elem})$. If the result of the map computation is computed by the surrounding fold expression whose function is $f_{fold}$, then the execution time of this fold expression is $m\cdot(\mathrm{T}\quad(f_{elem})+\mathrm{T}\quad(f_{fold}))$. If the fold expression does not use the result of a previous map operator, then its execution time is $m\cdot\mathrm{T}\quad(f_{fold})$.

## 4 Approximate Computation Synthesis

The Syndy's synthesis algorithm identifies several expression patterns in the original program and transforms them to produce a configurable approximate version of the program. The synthesized program exposes the set of configuration parameters that control the performance and accuracy of the computation.

**Program Configuration.** Each configuration parameter is a symbolic variable, which represents a probability of one probabilistic choice operator. We denote the configuration parameter as $q$. The configuration parameters take a value between 0 and 1. The program's *configuration vector* $\mathbf{c} = (q_1, \ldots, q_k)$ fully describes the available program approximation opportunities in a program $\mathcal{P}_\mathbf{c}$. In this section we present a set of rules that transform the original program $\mathcal{P}$ to produce the approximate program $\mathcal{P}\mathbf{c}$.

**Parameterized Probabilistic Choice.** We make a single extension of the language syntax to support the specification of the configurable approximate programs. We allow parameterized probabilistic operators, ${}_q\oplus$, which are controlled by the configuration variable $q$. This extends the previous definition that allowed only numerical constants.

However, this syntactic extension does not affect the semantics of the probabilistic choice operator. Instead, the programs with parameterized probabilistic choice operators replace the configuration variables with concrete execution probabilities at the beginning of the execution. Specifically, let a vector of numerical values $c = (n_1, \ldots, n_k)$

**1. Transformation of primitive function calls**

Original: $f(e_1, \ldots, e_k)$ inside an expression $e$
Synthesis: $f'_{\underline{q}}(x_1, \ldots, x_k) := f(x_1, \ldots, x_k)\ {}_{q_0}\oplus\ f_1(x_1, \ldots, x_k)\ {}_{q_1}\oplus \cdots\ {}_{q_{m-1}}\oplus\ f_m(x_1, \ldots, x_k)$
where $f_i$ are alternative available implementations provided in SpecF($f$)
Transformation: $e[f'_{\underline{q}}(e_1, \ldots, e_k)\ /\ f(e_1, \ldots, e_k)]$

**2. Transformation of map operators**

Original: $\mathbf{map}(f, \mathrm{L})$ where $f(x_1, \ldots, x_k) := e$
Synthesis: $f'_{\underline{q}}(x_1, \ldots, x_k) := e'_{\underline{q}}$
where $e'_{\underline{q}} = \mathrm{T}$ P F C $(e, \mathrm{SpecF})$
Transformation: $\mathbf{map}(f'_{\underline{q}}, \mathrm{L})$

**3. Transformation of sum fold operators**

Original: $\mathbf{fold}(0, plus, \mathrm{L})$ where $plus(a, b) := a + b$
Synthesis: $rect_q(x) := \frac{1}{q} \cdot x$
$plus'_q(a, b) := a + (rect_q(b)\ {}_q\oplus\ 0)$
Transformation: $\mathbf{fold}(0, plus'_q, \mathrm{L})$

**4. Transformation of maximum fold operators**

Original: $\mathbf{fold}(-\infty, max, \mathrm{L})$ where $max(a, b) := \max(a, b)$
Synthesis: $max'_q(a, b) := max(a, b)\ {}_q\oplus\ a$
Transformation: $\mathbf{fold}(n_{min}, max'_q, \mathrm{L})$

Fig. 7: Transformations of the Program

represent an *actual configuration* of the program $\mathcal{P}_{\mathbf{c}}$. The execution of the approximate program $\mathcal{P}_{\mathbf{c}}(c)$ starts by substituting each configuration parameter with corresponding probability, i.e., $\mathcal{P}_{\mathbf{c}}[n_1/q_1] \ldots [n_k/q_k]$. We discuss the synthesis of actual configuration parameters that provide profitable tradeoffs between performance and the accuracy of the transformed program in Section 5.

### 4.1 Computation Error

A computation error function specifies the expected difference between the results of the original and transformed program or a subprogram. We define several error functions for numerical expressions, lists of numerical expressions, and results of individual fold operators.

The error of an alternative expression $\hat{e}$ is an expected absolute difference, $\mathrm{Err}(e, \hat{e}) := \mathbb{E}[\,|e - \hat{e}|\,]$. The error of the alternative result of fold operators (except for maximization/minimization) is defined the same way. The error function of two numerical lists $L$ and $\hat{L}$ (which are typically the results of map operators) with multiplicity $m$ is defined as the maximum absolute error of its elements, $\mathrm{Err}(L, \hat{L}) := \max_i(\mathbb{E}[\,|L_i - \hat{L}_i|\,])$.

### 4.2 Computation Transformations

Figure 7 presents the synthesis and transformation rules for primitive functions, map operators, and summation and maximization fold operators. The algorithm constructs

new functions using specified synthesis rules and replaces the original expressions with the alternatives in the program body. The transformation of averaging and minimization operators are analogous to the summation and maximization cases.

In this section we present and discuss the expressions that characterize the error and performance of the synthesized operations. These expressions are a key prerequisite for the synthesis of profitable configurations. Specifically, we relate the derived expressions to the error and performance expressions used by the optimization framework from [21].

**Primitive Function Call Transformation.** Each primitive function call is replaced with call to a synthesized composite function $f'$ that contains a probabilistic choice operator between all alternative implementations specified in the set of the function's specification, A. We consider the case when the function $f$ has the original (most accurate) implementation $f_0$ and $m$ alternative (less accurate) implementations $f_1, \ldots, f_m$.

The set of symbolic variables $q_i$ ($0 \leq i \leq m$) comprise the configuration of the call site. The variable $q_0$ is the probability of executing $f_0$. Let $(E_i, T_i)$ be the error and time specifications for each alternative implementation $f_i$. Then, one can derive the expressions for the expected absolute error and execution time of this computations that follow from the definition of the expression error (using the fact that $\mathbb{E}[\,|e_1 - e_2 \;{}_p\!\oplus e_3|\,] = p\mathbb{E}[\,|e_1 - e_2|\,] + (1-p)\mathbb{E}[\,|e_1 - e_3|\,]$ and Eq. 1), and the expected execution time from the performance model from Section 3.4 ($\tau_\oplus$ is the time required to take one random sample):

$$E_g = q_0 \cdot 0 + q'_1 E_1 + q'_2 E_2 + \ldots + q'_m E_m \tag{4}$$

$$T_g = q'_0 T_0 + q'_1 T_1 + \ldots + q'_m T_m + m\tau_\oplus \tag{5}$$

In the previous expressions we used the helper variables $q'_i$. Each $q'_i$ is equal to the probabilities of executing the alternative implementation $f_i$. We relate $q'_i$ and the configuration variables $q_i$ as follows: $q'_0 = q_0$, each $q'_i = q_i \prod_{j=1}^{i-1}(1-q_j)$ for $i \in \{1, \ldots, m-1\}$, and $q'_m = 1 - \sum_{j=0}^{m-1} q_j$.

**Map Operator Transformation.** The synthesis algorithm replaces the composite function $f$ in a map operator with the synthesized function $f'$, in which all calls to the primitive functions are replaced by appropriate probabilistic choices between their alternative implementations (as described previously). The configuration vector of the function $f$ is a concatenation of the configuration vectors of all function calls that appear within the body of the function $f$.

We focus on specifying the total error of a single function call and the propagation of error induced by the previous computation. Let $f$ and $f'$ be the original and the alternate implementations. Then, from Eq. 3 total error induced by the computation is bounded by $\mathbb{E}[\,|f(e_1, \ldots, e_k) - \hat{f}(\hat{e}_1, \ldots, \hat{e}_k)|\,] \leq E_{f'} + \sum_{i=1}^{k} S_{f,i}\mathbb{E}[\,|e_i - \hat{e}_i|\,]$.

The error expressions of each function call within a map operator's function $f$ are the basis for computing the error that emerges and propagates through the body of $f$. Given these error expressions, one can use the approach presented in [21][Section 5.1] to compute the total error of the body of the function $f$. Specifically, the error propagation algorithm will multiply the error expression of each function call with the appropriate sensitivity indices of the functions that use the result of the call.

**Sum Fold Transformation.** The function *plus* that performs folding is replaced by an alternative that randomly computes the value of the input with probability $q$ or simply returns a previous temporary result with probability $1 - q$. To offset for the bias introduced by skipping some of the inputs, the sum fold operator performs *extrapolation*, by multiplying the sampled sum with $1/q$. The expected execution time is proportional to the $q$ fraction of the original node's execution time. Let $L = [e_1, \ldots, e_m]$. Then,

$$E_{\mathbf{sum}} = \mathbb{E}[\, |e_1 + \ldots + e_m - \tfrac{1}{q}((e_{1\ q} \oplus 0) + \ldots + (e_{m\ q} \oplus 0))| \,] \tag{6}$$

$$T_{\mathbf{sum}} = q \cdot \sum_{i=1}^{m} \mathrm{T}\quad(e_i) + m\tau_{\oplus} \tag{7}$$

When transforming the summation fold operator, the analysis checks if all inputs have the same error and time specifications, i.e., all subtrees of the computation are structurally isomorphic (the computation subtrees differ only in the numerical constants and the input variable names). A sufficient condition is if the computation that precedes the fold operator has only parameterized enumerated lists (or no enumerated lists).

If the inputs have the same error and time specifications, then one can derive an error expression equivalent to the one in [21], whose length does not depend on the size of the input list. Therefore, the analysis can immediately use the solver from [21]. This includes the computation of the expected execution time: if the expected execution time of an input is $\tau_e$, then $T_{\mathbf{sum}} = q \cdot m \cdot \tau_e + m\tau_{\oplus}$. If the inputs have different error and time specifications, the analysis unfolds the computation of the fold operator. We discuss this part of the analysis in Section 5.

**Maximization Fold Transformation.** The function *max* that performs folding is replaced by an alternative implementation that randomly selects and computes the value of the input with probability $q$ or returns a previous neutral temporary result. The statically computed initial value $c_{min}$ is the minimum value that the elements in $L$ can evaluate to.

The definition of error for the maximization operator differs from the expected absolute error. Instead, we define the error for the maximization operator as a *percentile error*. The percentile error is 0 if the result of the approximate fold computation is one of the top few ordered elements; otherwise the penalty is proportional to the case when the approximate fold operator returns the (worst) minimum element of the input list. Percentile error is appropriate in a scenario which does not require strictly returning the maximum element, and instead returning one of the top $\alpha = \lfloor \kappa \cdot m \rfloor$ elements incurs no penalty (for a small constant $\kappa$ that specifies the fraction of top ordered elements that do not incur error).

The error function $E_{\mathbf{min}} = \mathrm{Err}(\mathrm{L}, \hat{e})$ takes as input the list of the inputs of the original fold operator $L = [e_1, \ldots, e_m]$ and the fold operator $\hat{e} = \mathbf{fold}_q(0, max'_q, \mathrm{L})$. The analysis of the maximization fold operator requires (like the analysis of summation) that the expressions $e_1$ to $e_m$ be structurally isomorphic, to bound the propagated error with an expression whose length does not depend on the size of the input list.

We define the error function as follows. Let each $e_i$ evaluate to $n_i$ and we define the auxiliary set $T(\hat{n}) = \{n_i \ : \ n_i > \hat{n}\}$ to contain the input list elements that are greater than the result $\hat{n}$ returned by the approximate fold operator. Note that $\hat{n}$ is always equal

to one of $n_1, \ldots, n_m$. Furthermore, let $B$ be the maximum absolute distance between the two inputs $n_i$ and $n_j$. Then the error function for a particular $\hat{n}$ is

$$\mathcal{E}(\hat{n}) = \begin{cases} 0, & \text{if } |T(\hat{n})| \leq \alpha \\ B, & \text{otherwise} \end{cases} \tag{8}$$

If the number of the elements of the map operator that are smaller than $\hat{n}$ (the value that the fold operator produces) is at most $\kappa \cdot m$, then the computation does not incur penalty and the error is 0. Otherwise, the error is the maximum absolute difference between any of the results of the map operator and the result of the fold operator. This difference can be bounded by the difference between the largest and the smallest elements from the input list.

The expected error is then $E_{min} = \mathbb{E}[\,\mathcal{E}(\hat{n})\,]$. The probability that the approximate maximization operator returns a specific value $\hat{n}$ is a function of the probability $q$. One can derive that the expected error is proportional to the probability that the value $\hat{n}$ is not among the maximum $\alpha$ elements, $c \cdot \binom{m-\alpha}{q \cdot m} / \binom{m}{q \cdot m}$, where $c$ is a known constant. The expected time of the maximization fold operator becomes, like for the sum operator, $T_{\mathbf{max}} = q \cdot m \cdot \tau_e + m\tau_{\oplus}$.

## 5 Optimal Configuration Synthesis

Once it generates the approximate randomized computation with exposed configuration parameters, Syndy translates the map fold program (with the transformation locations) to the accuracy-aware model of computation and uses the optimization algorithm presented in [21] to search for tradeoffs. In this section we outline the translation procedure.

### 5.1 Program Translation to Accuracy-aware Model

**Primitive Function Call Translation.** Each primitive function is translated to a function node. The specification of the sensitivity of a function node and the specification of alternative implementations is the same as the specification described in this paper.

**Arithmetic Operation Translation.** Arithmetic operations are also represented using function nodes. The addition and subtraction operations map into a function node with a single implementation and sensitivity indices $S_1 = S_2 = 1$. The sensitivity of the multiplication operation, however, depends on the intervals of the input parameters. Overall, the sensitivity $S_1$ is equal to the maximum absolute value of the other expression $e_2$. Syndy uses a standard interval analysis to compute this maximum value. Therefore, each multiplication operation will be translated to a new function node, each with a unique sensitivity specification, but none of these implementations incur accuracy loss.

**Probabilistic Choice Translation.** A probabilistic choice operator $e_1 \,{}_p\!\oplus\, e_2$ with a constant probability $p$ is translated to a function node whose sensitivity is equal to the maximum of the sensitivities of its arguments and it induces the expected error equal to $2p(1-p)\mathbb{E}[\,|e_1 - e_2|\,]$ for deterministic $e_1$ and $e_2$. Its execution time is equal to $\tau_{\oplus}$. In n addition, the execution times of the previous computation $e_1$ and $e_2$ are multiplied by $p$ and $1-p$, respectively.

**Composite Function Call Translation.** A dataflow graph of each composite function $f(x_1, \ldots, x_k) := e_f$ has the form of a direct acyclic graph (since the computation does not support loops or recursion). The composite function body $e_f$ is translated to a directed acyclic graph that represents the data flow in the function expression. Each node represents a result of a called primitive function or a basic operation. Each edge represents the data flow from the node that produces a value to a node that uses this value. The translation algorithm inlines all called composite functions and thus operates on an extended dataflow graph.

**Map Operator Translation.** We consider the two cases for translating map operators:

- *Program root or followed by a fold operator.* A map operator is translated to a computation node. The algorithms constructs the dataflow graph for the function called by the map operator. If the function is primitive, the dataflow graph consists of only a single node. The dataflow graph of composite functions is computed as we described previously. The multiplicity of the map operator is equal to the size of its input list.
- *Sequence of map operators.* If the output of one map operator is passed as an input to another map operator, then such a sequence is first transformed to a single map operator whose function encompasses the functions of the original map operators. The new map operator is translated to an equivalent computation node as outlined above. For instance, the subprogram $\mathbf{map}(f_1, \mathbf{map}(f_2, L))$ is transformed to $\mathbf{map}(f', L)$ where the new composite function $f'(x) := f_1(f_2(x))$. The new map operator is then translated to an equivalent computation node.

The computation node's inputs are connected to nodes that produce these inputs or the program's inputs. The computation node's output is connected to nodes that use the results of this computation node. The multiplicity of the node is derived from the map operator's input list specification.

**Fold Operator Translation.** The translation first identifies if the fold operator belongs to the class of analyzable functions (i.e., summation or maximization) and the input list contains isomorphic elements. If both conditions are true, then the fold operator is translated to the reduction node. Otherwise, it is translated to the computation node:

- *Translation to Reduction Node.* The translation constructs a special reduction node for the fold operator, based on the recognized function. This reduction node is connected with an edge to the previous computation node that represents map operator that computes the input list of the fold operator. The multiplicity of the node, $m$, is obtained from the program's input specification.
- *Translation to Computation Node.* The fold operator with the function $f$ is unfolded $m$ times. If the function $f$ is recognized, it is replaced with the transformed version from Figure 7. The input of each of the fold operator's function calls is connected with an edges to 1) the previous call to the function $f$ and 2) to the function node that represents the computation producing one of the inputs. The translation constructs the computation node containing these function nodes. The multiplicity of this computation node is 1.

The fold operator's output is connected to the nodes representing the computation that use the fold operator's results.

### 5.2 Transformation Properties

We first compare the error expressions derived for the transformed expressions in Section 4 and the error expressions for the nodes in the model of computation from [21].

**Theorem 1.** *The error of the Syndy's expressions is bounded by expression* $c \cdot E_{model}$ *where* $E_{model}$ *is the error expression defined for the appropriate node in the accuracy-aware model of computation and* $c$ *is a known constant.*

To prove this theorem, we continue the derivation of the error expressions for the operators and match the derived terms with the errors of the nodes in the computational model presented in [21]. This result allows us to use this optimization framework to search for the optimal tradeoffs and provide guarantees that the error of the transformed program is bounded by the error specified on the tradeoff curve.

For each set of applied transformations, there exists a configuration of the program that produces the same result as the original (fully accurate) implementation of the computation:

**Theorem 2.** *Let* $\mathcal{P}$ *be the original deterministic program and* $\mathcal{P}_c$ *an approximate program synthesized using transformations from Figure 7. If* $q_0 = 1$ *for all primitive function configurations and* $q = 1$ *for all fold configurations, then* $\mathbb{E}[\,|\mathcal{P} - \mathcal{P}_c|\,] = 0$.

The proof follows by induction on the expressions in the transformed program. Specifically, for each expression one can find that setting the configuration to execute the original implementation yields no error.

### 5.3 Optimal Parameter Search

The parameter search algorithm takes the model graph produced by the translation algorithm and the specification of the alternative computations. The search algorithm uses constrained mathematical optimization to find the configuration of the transformed model that executes in a minimum amount of time while satisfying a specific error bound [21]. The algorithm is guaranteed to find a $(1 + \varepsilon)$-approximation of the globally minimal execution time for a given error bound. The user-settable accuracy parameter $\varepsilon$ determines how close the computed optimum time/error is to the global optimum.

The parameter search algorithm produces a tradeoff curve between the expected absolute error and the expected execution time. Each point on the tradeoff curve contains the actual configuration of the approximate program. This tradeoff curve can be used to instantiate (as described in Section 4) the transformed program $\mathcal{P}_{\mathbf{c}}(c)$ with the actual configuration $c$. Specifically, if a user specifies the program error bound $\Delta$, the configuration $c$ is associated with the point on the tradeoff curve that corresponds to the bound $\Delta$. The search algorithm guarantees that the expected absolute error of the approximate program is smaller than $\Delta$ (i.e., $\mathbb{E}[\,|\mathcal{P} - \mathcal{P}_{\mathbf{c}}(c)|\,] \leq \Delta$) for the configuration $c$.

**Implementation.** We implemented a prototype of Syndy's synthesis framework, including the algorithm for finding optimal configurations. The implementation consists of 2000 lines of C++ code and 1300 lines of OCaml code. The optimization algorithm uses the lp_solve [1] linear programming library.

| Program | Functions | Map/Fold | Configurations | Time (s) |
|---|---|---|---|---|
| btrack | 8 | 1/1 | 142 | 1.58 |
| bscholes | 6 | 2/1 | 5259 | 1.73 |
| integral | 6 | 1/1 | 352 | 0.36 |
| mlikelihood | 10 | 3/3 | 999 | 18.93 |

Fig. 8: Result Summary

## 6 Evaluation

We perform the initial evaluation on four map-fold computations: 1) btrack is the example from Section 2, 2) bscholes computes Black-Scholes formula for a list of stock options, 3) integral computes a numerical computation given as the example in [21], and 4) mlikelihood is a maximum likelihood computation that aggregates the results of summation computation that sums two parts of the feature vector. The optimization constant is $\varepsilon = 0.01$, instructing the algorithm to produce a tradeoff that is within 1% of the optimal tradeoff. We performed the evaluation on 2.2 GHz Intel Xeon E5520 with 16 GB of main memory.

Table 8 present the results of Syndy's evaluation. It presents the number of alternative function implementations (equal to the size of the configuration vector for a map operator) and the number of map and fold operators. For each benchmark we present the number of configurations of the computation selected by optimization algorithm and the execution time of the algorithm.

All benchmark analyses completed in within 20 s. We note that the execution time does not depend on the size of the input lists, but only on the number of alternative functions and the number of map and fold operators. The majority of the analysis execution time is spent in the finding the optimum solution for the fold operators. This search procedure computes intermediate solutions for a large number of sections of the tradeoff curve. On the other hand, the optimization of the map computation is performed by solving constructed linear programs whose size is from several to several hundreds of variables. Therefore, the mlikelihood benchmark, which has 3 fold operations, consumes more time for analysis than the benchmarks with only a single fold operator.

## 7 Related Work

**Quantitative Program Synthesis.** Researchers have recently explored techniques for the automatic generation of optimized programs that operate with variable accuracy using numerical optimization techniques, primarily for reactive control systems.

Smooth interpretation [10,9] uses a gradient descent based method to synthesize control parameters of imperative computer programs. The analysis returns a set of parameters that minimize the difference between the expected and computed control values for controllers. Our paper, in contrast, presents a technique that, given a specification of input intervals, produces a set of configurations that explicitly trade accuracy and performance and provides guarantees on the induced error using constrained optimization techniques.

Quasy [7] explores tradeoffs between quantitative properties of the system such as the execution time, energy, or accuracy of control systems. Quasy uses linear temporal logic to reason about the constraints and represents the underlying state as a Markov chain. Von Essen and Jobstmann [20] present a quantitative model checking framework for expressing general tradeoffs between objectives specified in temporal logic by representing problems as Markov Decision Processes. Our paper presents an algorithm for map-fold computations, for which it produces a whole set of profitable tradeoffs between accuracy and performance.

**Accuracy Analysis of Program Transformations.** Researchers have presented several papers on static analysis of program transformations that affect accuracy of results. Misailovic et al. [13] present assume-guarantee probabilistic analyses for several loop patterns amenable to loop perforation. The values that the computations operate on are represented as random variables. Chaudhuri et al. [8] present an analysis of additional pattern for loops amenable to loop perforation and use the analysis of Lipschitz continuity of functions to provide probabilistic bounds for perforatable loops. In the differential privacy context, Pierce and Reed [15] present a type system that ensures that a noise added to preserve privacy does not significantly affect the result of reduction operations. Barthe et al. [4] present a relational probabilistic framework to reason about arbitrary differential privacy mechanisms. Researchers have also recently explored techniques based on symbolic execution to check the probability of satisfaction of program's assertions [11,18].

These techniques treat some of the values that the computations operates on as random quantities and quantify the effects of these random quantities on the accuracy of the result computations produce. However, the main focus of these techniques is the analysis of error and they do not search for optimal tradeoffs between the introduced error and performance of the applications.

## 8 Conclusion

The field of program optimization has focused, almost exclusively since the inception of the field, on transformations that do not change the result that the computation produces. The recent emergence of approximate program transformation and synthesis algorithms promises to dramatically increase the scope and relevance of program optimization techniques in a world increasingly dominated by computations that can profitably trade off accuracy in return for increased performance. Syndy provides an opportunity to exploit such profitable tradeoffs by automatically transforming a set of computations, while proving optimal probabilistic guarantees for the accuracy of the computation for a whole computation.

## References


1. lp_solve linear programming library. http://lpsolve.sourceforge.net/.
2. J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. PLDI, 2009.
3. W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. PLDI, 2010.

4. G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic reasoning for differential privacy. POPL, 2012.
5. C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*.
6. M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.
7. K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. Quasy: Quantitative synthesis tool. In *TACAS*, 2011.
8. S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. FSE, 2011.
9. S. Chaudhuri and A. Solar-Lezama. Smoothing a program soundly and robustly. In *CAV'11*.
10. S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *PLDI*, 2010.
11. A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. ICSE, 2013.
12. H. Hoffman, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS, 2011.
13. S. Misailovic, D. Roy, and M. Rinard. Probabilistically Accurate Program Transformations. SAS, 2011.
14. S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, 2010.
15. J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *ICFP*, 2010.
16. M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.
17. M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. OOPSLA, 2007.
18. S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, 2013.
19. S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. FSE '11.
20. C. Von Essen and B. Jobstmann. Synthesizing efficient controllers. In *VMCAI*, 2012.
21. Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. POPL, 2012.